# Chapter 2
# Intelligent Agents

*In which we discuss the nature of agents, perfect or otherwise, the diversity of environments, and the resulting menagerie of agent types.*

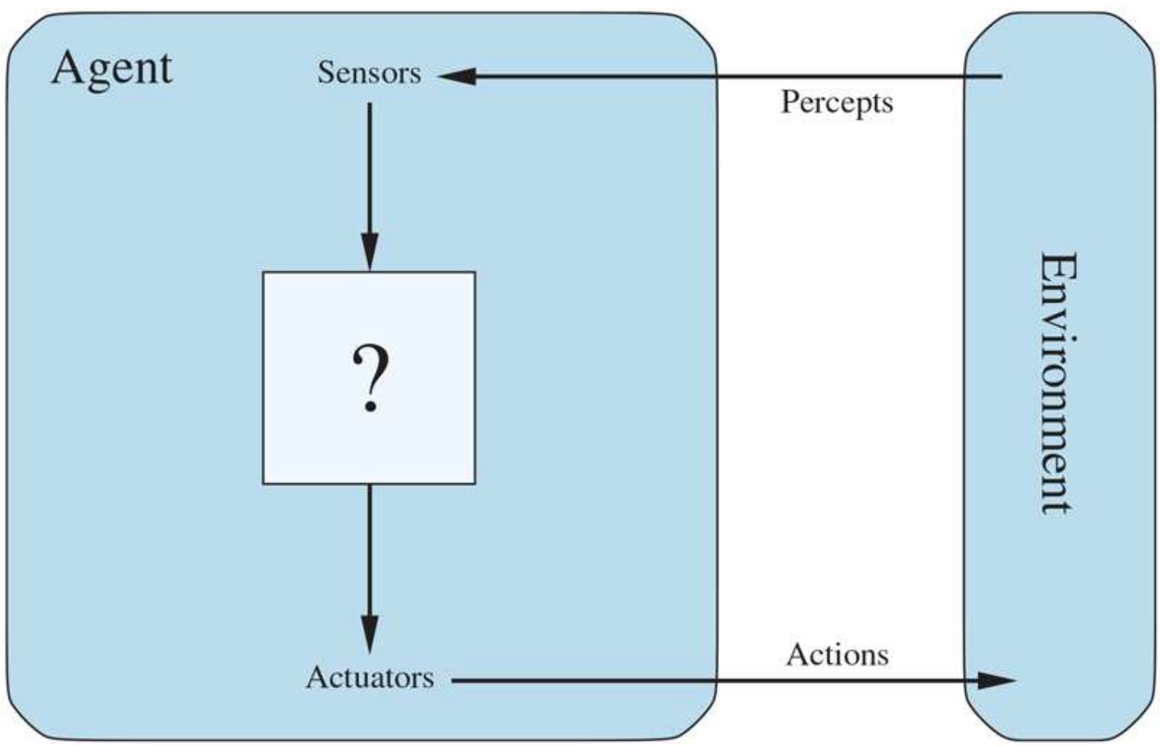
Chapter 1 identified the concept of **rational agents** as central to our approach to artificial intelligence. In this chapter, we make this notion more concrete. We will see that the concept of rationality can be applied to a wide variety of agents operating in any imaginable environment. Our plan in this book is to use this concept to develop a small set of design principles for building successful agents—systems that can reasonably be called **intelligent**.

We begin by examining agents, environments, and the coupling between them. The observation that some agents behave better than others leads naturally to the idea of a rational agent—one that behaves as well as possible. How well an agent can behave depends on the nature of the environment; some environments are more difficult than others. We give a crude categorization of environments and show how properties of an environment influence the design of suitable agents for that environment. We describe a number of basic "skeleton" agent designs, which we flesh out in the rest of the book.

## 2.1 Agents and Environments

An **agent** is anything that can be viewed as perceiving its **environment** through **sensors** and acting upon that environment through **actuators**. This simple idea is illustrated in Figure 2.1. A human agent has eyes, ears, and other organs for sensors and hands, legs, vocal tract, and so on for actuators. A robotic agent might have cameras and infrared range finders for sensors and various motors for actuators. A software agent receives file contents, network packets, and human input (keyboard/mouse/touchscreen/voice) as sensory inputs and acts on the environment by writing files, sending network packets, and displaying information or generating sounds. The environment could be everything—the entire universe! In practice it is just that part of the universe whose state we care about when designing this agent—the part that affects what the agent perceives and that is affected by the agent's actions.

Figure 2.1




Agents interact with environments through sensors and actuators.

*Environment*

*Sensor*

*Actuator*

We use the term **percept** to refer to the content an agent's sensors are perceiving. An agent's **percept sequence** is the complete history of everything the agent has ever perceived. In general, *an agent's choice of action at any given instant can depend on its built-in knowledge and on the entire percept sequence observed to date, but not on anything it hasn't perceived.* By specifying the agent's choice of action for every possible percept sequence, we have said more or less everything there is to say about the agent. Mathematically speaking, we say that an agent's behavior is described by the **agent function** that maps any given percept sequence to an action.

*Percept*

*Percept sequence*

*Agent function*

We can imagine *tabulating* the agent function that describes any given agent; for most agents, this would be a very large table—infinite, in fact, unless we place a bound on the length of percept sequences we want to consider. Given an agent to experiment with, we can, in principle, construct this table by trying out all possible percept sequences and recording which actions the agent does in response.[1] The table is, of course, an *external* characterization of the agent. *Internally*, the agent function for an artificial agent will be implemented by an **agent program**. It is important to keep these two ideas distinct. The agent function is an abstract mathematical description; the agent program is a concrete implementation, running within some physical system.

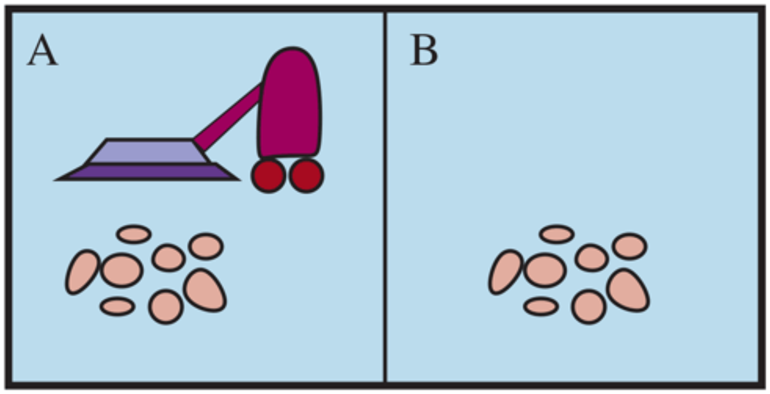
1 If the agent uses some randomization to choose its actions, then we would have to try each sequence many times to identify the probability of each action. One might imagine that acting randomly is rather silly, but we show later in this chapter that it can be very intelligent.

*Agent program*

To illustrate these ideas, we use a simple example—the vacuum-cleaner world, which consists of a robotic vacuum-cleaning agent in a world consisting of squares that can be either dirty or clean. Figure 2.2 shows a configuration with just two squares, $A$ and $B$. The vacuum agent perceives which square it is in and whether there is dirt in the square. The agent starts in square $A$. The available actions are to move to the right, move to the left, suck up the dirt, or do nothing.[2] One very simple agent function is the following: if the current square is dirty, then suck; otherwise, move to the other square. A partial tabulation of this agent function is shown in Figure 2.3 and an agent program that implements it appears in Figure 2.8 on page 49.

2 In a real robot, it would be unlikely to have an actions like "move right" and "move left." Instead the actions would be "spin wheels forward" and "spin wheels backward." We have chosen the actions to be easier to follow on the page, not for ease of implementation in an actual robot.

Figure 2.2




A vacuum-cleaner world with just two locations. Each location can be clean or dirty, and the agent can move left or right and can clean the square that it occupies. Different versions of the vacuum world allow for different rules about what the agent can perceive, whether its actions always succeed, and so on.

---

Figure 2.3

| Percept sequence | Action |
|---|---|
| $[A, Clean]$ | *Right* |
| $[A, Dirty]$ | *Suck* |
| $[B, Clean]$ | *Left* |
| $[B, Dirty]$ | *Suck* |
| $[A, Clean], [A, Clean]$ | *Right* |
| $[A, Clean], [A, Dirty]$ | *Suck* |
| $\vdots$ | $\vdots$ |
| $[A, Clean], [A, Clean], [A, Clean]$ | *Right* |
| $[A, Clean], [A, Clean], [A, Dirty]$ | *Suck* |
| $\vdots$ | $\vdots$ |

Partial tabulation of a simple agent function for the vacuum-cleaner world shown in Figure 2.2. The agent cleans the current square if it is dirty, otherwise it moves to the other square. Note that the table is of unbounded size unless there is a restriction on the length of possible percept sequences.

---

Looking at Figure 2.3, we see that various vacuum-world agents can be defined simply by filling in the right-hand column in various ways. The obvious question, then, is this: *What is the right way to fill out the table?* In other words, what makes an agent good or bad, intelligent or stupid? We answer these questions in the next section.

Before closing this section, we should emphasize that the notion of an agent is meant to be a tool for analyzing systems, not an absolute characterization that divides the world into agents and non-agents. One could view a hand-held calculator as an agent that chooses the action of displaying “4” when given the percept sequence “$2 + 2 =$ ,” but such an analysis would hardly aid our understanding of the calculator. In a sense, all areas of engineering can be seen as designing artifacts that interact with the world; AI operates at (what the authors consider to be) the most interesting end of the spectrum, where the artifacts have significant computational resources and the task environment requires nontrivial decision making.

## 2.2 Good Behavior: The Concept of Rationality

A **rational agent** is one that does the right thing. Obviously, doing the right thing is better than doing the wrong thing, but what does it mean to do the right thing?

*Rational agent*

### 2.2.1 Performance measures

Moral philosophy has developed several different notions of the "right thing," but AI has generally stuck to one notion called **consequentialism**: we evaluate an agent's behavior by its consequences. When an agent is plunked down in an environment, it generates a sequence of actions according to the percepts it receives. This sequence of actions causes the environment to go through a sequence of states. If the sequence is desirable, then the agent has performed well. This notion of desirability is captured by a **performance measure** that evaluates any given sequence of environment states.

*Consequentialism*

*Performance measure*

Humans have desires and preferences of their own, so the notion of rationality as applied to humans has to do with their success in choosing actions that produce sequences of environment states that are desirable *from their point of view*. Machines, on the other hand, do *not* have desires and preferences of their own; the performance measure is, initially at least, in the mind of the designer of the machine, or in the mind of the users the machine is designed for. We will see that some agent designs have an explicit representation of (a version of) the performance measure, while in other designs the performance measure is entirely implicit—the agent may do the right thing, but it doesn't know why.

Recalling Norbert Wiener's warning to ensure that "the purpose put into the machine is the purpose which we really desire" (page 33), notice that it can be quite hard to formulate a performance measure correctly. Consider, for example, the vacuum-cleaner agent from the preceding section. We might propose to measure performance by the amount of dirt cleaned up in a single eight-hour shift. With a rational agent, of course, what you ask for is what you get. A rational agent can maximize this performance measure by cleaning up the dirt, then dumping it all on the floor, then cleaning it up again, and so on. A more suitable performance measure would reward the agent for having a clean floor. For example, one point could be awarded for each clean square at each time step (perhaps with a penalty for electricity consumed and noise generated). *As a general rule, it is better to design performance measures according to what one actually wants to be achieved in the environment, rather than according to how one thinks the agent should behave.*

Even when the obvious pitfalls are avoided, some knotty problems remain. For example, the notion of "clean floor" in the preceding paragraph is based on average cleanliness over time. Yet the same average cleanliness can be achieved by two different agents, one of which does a mediocre job all the time while the other cleans energetically but takes long breaks. Which is preferable might seem to be a fine point of janitorial science, but in fact it is a deep philosophical question with far-reaching implications. Which is better—a reckless life of highs and lows, or a safe but humdrum existence? Which is better—an economy where everyone lives in moderate poverty, or one in which some live in plenty while others are very poor? We leave these questions as an exercise for the diligent reader.

For most of the book, we will assume that the performance measure can be specified correctly. For the reasons given above, however, we must accept the possibility that we might put the wrong purpose into the machine—precisely the King Midas problem described on page 33. Moreover, when designing one piece of software, copies of which will belong to different users, we cannot anticipate the exact preferences of each individual user. Thus, we may need to build agents that reflect initial uncertainty about the true performance measure and learn more about it as time goes by; such agents are described in Chapters 16, 18, and 22.

## 2.2.2 Rationality

What is rational at any given time depends on four things:

- The performance measure that defines the criterion of success.
- The agent's prior knowledge of the environment.
- The actions that the agent can perform.
- The agent's percept sequence to date.

This leads to a **definition of a rational agent**:

> For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

*Definition of a rational agent*

Consider the simple vacuum-cleaner agent that cleans a square if it is dirty and moves to the other square if not; this is the agent function tabulated in Figure 2.3. Is this a rational agent? That depends! First, we need to say what the performance measure is, what is known about the environment, and what sensors and actuators the agent has. Let us assume the following:

- The performance measure awards one point for each clean square at each time step, over a "lifetime" of 1000 time steps.
- The "geography" of the environment is known *a priori* (Figure 2.2) but the dirt distribution and the initial location of the agent are not. Clean squares stay clean and sucking cleans the current square. The $Right$ and $Left$ actions move the agent one square except when this would take the agent outside the environment, in which case the agent remains where it is.
- The only available actions are $Right$, $Left$, and $Suck$.
- The agent correctly perceives its location and whether that location contains dirt.

Under these circumstances the agent is indeed rational; its expected performance is at least as good as any other agent's.

One can see easily that the same agent would be irrational under different circumstances. For example, once all the dirt is cleaned up, the agent will oscillate needlessly back and forth; if the performance measure includes a penalty of one point for each movement, the agent will fare poorly. A better agent for this case would do nothing once it is sure that all the squares are clean. If clean squares can become dirty again, the agent should occasionally check and re-clean them if needed. If the geography of the environment is unknown, the agent will need to **explore** it. Exercise 2.VACR asks you to design agents for these cases.

### 2.2.3 Omniscience, learning, and autonomy

*Omniscience*

We need to be careful to distinguish between rationality and **omniscience**. An omniscient agent knows the *actual* outcome of its actions and can act accordingly; but omniscience is impossible in reality. Consider the following example: I am walking along the Champs Elysées one day and I see an old friend across the street. There is no traffic nearby and I'm not otherwise engaged, so, being rational, I start to cross the street. Meanwhile, at 33,000 feet, a cargo door falls off a passing airliner,[3] and before I make it to the other side of the street I am flattened. Was I irrational to cross the street? It is unlikely that my obituary would read "Idiot attempts to cross street."

3 See N. Henderson, "New door latches urged for Boeing 747 jumbo jets," *Washington Post*, August 24, 1989.

This example shows that rationality is not the same as perfection. Rationality maximizes *expected* performance, while perfection maximizes *actual* performance. Retreating from a requirement of perfection is not just a question of being fair to agents. The point is that if we expect an agent to do what turns out after the fact to be the best action, it will be impossible to design an agent to fulfill this specification—unless we improve the performance of crystal balls or time machines.

Our definition of rationality does not require omniscience, then, because the rational choice depends only on the percept sequence *to date*. We must also ensure that we haven't inadvertently allowed the agent to engage in decidedly underintelligent activities. For example, if an agent does not look both ways before crossing a busy road, then its percept sequence will not tell it that there is a large truck approaching at high speed. Does our definition of rationality say that it's now OK to cross the road? Far from it!

First, it would not be rational to cross the road given this uninformative percept sequence: the risk of accident from crossing without looking is too great. Second, a rational agent should choose the "looking" action before stepping into the street, because looking helps maximize the expected performance. Doing actions *in order to modify future percepts*—sometimes called **information gathering**—is an important part of rationality and is covered in depth in Chapter 16. A second example of information gathering is provided by the **exploration** that must be undertaken by a vacuum-cleaning agent in an initially unknown environment.

---

*Information gathering*

Our definition requires a rational agent not only to gather information but also to **learn** as much as possible from what it perceives. The agent's initial configuration could reflect some prior knowledge of the environment, but as the agent gains experience this may be modified and augmented. There are extreme cases in which the environment is completely known *a priori* and completely predictable. In such cases, the agent need not perceive or learn; it simply acts correctly.

---

*Learning*

Of course, such agents are fragile. Consider the lowly dung beetle. After digging its nest and laying its eggs, it fetches a ball of dung from a nearby heap to plug the entrance. If the ball of dung is removed from its grasp *en route*, the beetle continues its task and pantomimes plugging the nest with the nonexistent dung ball, never noticing that it is missing. Evolution has built an assumption into the beetle's behavior, and when it is violated, unsuccessful behavior results.

Slightly more intelligent is the sphex wasp. The female sphex will dig a burrow, go out and sting a caterpillar and drag it to the burrow, enter the burrow again to check all is well, drag the caterpillar inside, and lay its eggs. The caterpillar serves as a food source when the eggs hatch. So far so good, but if an entomologist moves the caterpillar a few inches away while the sphex is doing the check, it will revert to the "drag the caterpillar" step of its plan and will continue the plan without modification, re-checking the burrow, even after dozens of caterpillar-moving interventions. The sphex is unable to learn that its innate plan is failing, and thus will not change it.

To the extent that an agent relies on the prior knowledge of its designer rather than on its own percepts and learning processes, we say that the agent lacks **autonomy**. A rational agent should be autonomous—it should learn what it can to compensate for partial or incorrect prior knowledge. For example, a vacuum-cleaning agent that learns to predict where and when additional dirt will appear will do better than one that does not.

*Autonomy*

As a practical matter, one seldom requires complete autonomy from the start: when the agent has had little or no experience, it would have to act randomly unless the designer gave some assistance. Just as evolution provides animals with enough built-in reflexes to survive long enough to learn for themselves, it would be reasonable to provide an artificial intelligent agent with some initial knowledge as well as an ability to learn. After sufficient experience of its environment, the behavior of a rational agent can become effectively *independent* of its prior knowledge. Hence, the incorporation of learning allows one to design a single rational agent that will succeed in a vast variety of environments.

## 2.3 The Nature of Environments

Now that we have a definition of rationality, we are almost ready to think about building rational agents. First, however, we must think about **task environments**, which are essentially the "problems" to which rational agents are the "solutions." We begin by showing how to specify a task environment, illustrating the process with a number of examples. We then show that task environments come in a variety of flavors. The nature of the task environment directly affects the appropriate design for the agent program.

---

*Task environment*

### 2.3.1 Specifying the task environment

In our discussion of the rationality of the simple vacuum-cleaner agent, we had to specify the performance measure, the environment, and the agent's actuators and sensors. We group all these under the heading of the **task environment**. For the acronymically minded, we call this the **PEAS** (**P**erformance, **E**nvironment, **A**ctuators, **S**ensors) description. In designing an agent, the first step must always be to specify the task environment as fully as possible.

---

*PEAS*

The vacuum world was a simple example; let us consider a more complex problem: an automated taxi driver. Figure 2.4 summarizes the PEAS description for the taxi's task environment. We discuss each element in more detail in the following paragraphs.

---

Figure 2.4

---

| Agent Type | Performance Measure | Environment | Actuators | Sensors |
|---|---|---|---|---|
| Taxi driver | Safe, fast, legal, comfortable trip, maximize profits, minimize impact on other road users | Roads, other traffic, police, pedestrians, customers, weather | Steering, accelerator, brake, signal, horn, display, speech | Cameras, radar, speedometer, GPS, engine sensors, accelerometer, microphones, touchscreen |

PEAS description of the task environment for an automated taxi driver.

First, what is the **performance measure** to which we would like our automated driver to aspire? Desirable qualities include getting to the correct destination; minimizing fuel consumption and wear and tear; minimizing the trip time or cost; minimizing violations of traffic laws and disturbances to other drivers; maximizing safety and passenger comfort; maximizing profits. Obviously, some of these goals conflict, so tradeoffs will be required.

Next, what is the driving **environment** that the taxi will face? Any taxi driver must deal with a variety of roads, ranging from rural lanes and urban alleys to 12-lane freeways. The roads contain other traffic, pedestrians, stray animals, road works, police cars, puddles, and potholes. The taxi must also interact with potential and actual passengers. There are also some optional choices. The taxi might need to operate in Southern California, where snow is seldom a problem, or in Alaska, where it seldom is not. It could always be driving on the right, or we might want it to be flexible enough to drive on the left when in Britain or Japan. Obviously, the more restricted the environment, the easier the design problem.

The **actuators** for an automated taxi include those available to a human driver: control over the engine through the accelerator and control over steering and braking. In addition, it will need output to a display screen or voice synthesizer to talk back to the passengers, and perhaps some way to communicate with other vehicles, politely or otherwise.

The basic **sensors** for the taxi will include one or more video cameras so that it can see, as well as lidar and ultrasound sensors to detect distances to other cars and obstacles. To avoid speeding tickets, the taxi should have a speedometer, and to control the vehicle properly, especially on curves, it should have an accelerometer. To determine the mechanical state of the vehicle, it will need the usual array of engine, fuel, and electrical system sensors. Like many human drivers, it might want to access GPS signals so that it doesn't get lost. Finally, it will need touchscreen or voice input for the passenger to request a destination.

In Figure 2.5, we have sketched the basic PEAS elements for a number of additional agent types. Further examples appear in Exercise 2.PEAS. The examples include physical as well as virtual environments. Note that virtual task environments can be just as complex as the "real" world: for example, a **software agent** (or software robot or **softbot**) that trades on auction and reselling Web sites deals with millions of other users and billions of objects, many with real images.

Figure 2.5

| Agent Type | Performance Measure | Environment | Actuators | Sensors |
|---|---|---|---|---|
| Medical diagnosis system | Healthy patient, reduced costs | Patient, hospital, staff | Display of questions, tests, diagnoses, treatments | Touchscreen/voice entry of symptoms and findings |
| Satellite image analysis system | Correct categorization of objects, terrain | Orbiting satellite, downlink, weather | Display of scene categorization | High-resolution digital camera |
| Part-picking robot | Percentage of parts in correct bins | Conveyor belt with parts; bins | Jointed arm and hand | Camera, tactile and joint angle sensors |
| Refinery controller | Purity, yield, safety | Refinery, raw materials, operators | Valves, pumps, heaters, stirrers, displays | Temperature, pressure, flow, chemical sensors |
| Interactive English tutor | Student's score on test | Set of students, testing agency | Display of exercises, feedback, speech | Keyboard entry, voice |

Examples of agent types and their PEAS descriptions.

*Software agent*

*Softbot*

## 2.3.2 Properties of task environments

The range of task environments that might arise in AI is obviously vast. We can, however, identify a fairly small number of dimensions along which task environments can be categorized. These dimensions determine, to a large extent, the appropriate agent design and the applicability of each of the principal families of techniques for agent implementation. First we list the dimensions, then we analyze several task environments to illustrate the ideas. The definitions here are informal; later chapters provide more precise statements and examples of each kind of environment.

*Fully observable*

*Partially observable*

**FULLY OBSERVABLE VS. PARTIALLY OBSERVABLE:** If an agent's sensors give it access to the complete state of the environment at each point in time, then we say that the task environment is fully observable. A task environment is effectively fully observable if the sensors detect all aspects that are *relevant* to the choice of action; relevance, in turn, depends on the performance measure. Fully observable environments are convenient because the agent need not maintain any internal state to keep track of the world. An environment might be partially observable because of noisy and inaccurate sensors or because parts of the state are simply missing from the sensor data—for example, a vacuum agent with only a local dirt sensor cannot tell whether there is dirt in other squares, and an automated taxi cannot see what other drivers are thinking. If the agent has no sensors at all then the environment is **unobservable**. One might think that in such cases the agent's plight is hopeless, but, as we discuss in Chapter 4, the agent's goals may still be achievable, sometimes with certainty.

*Unobservable*

**SINGLE-AGENT VS. MULTIAGENT:** The distinction between single-agent and multiagent environments may seem simple enough. For example, an agent solving a crossword puzzle by itself is clearly in a single-agent environment, whereas an agent playing chess is in a two-agent environment. However, there are some subtle issues. First, we have described how an entity *may* be viewed as an agent, but we have not explained which entities *must* be viewed as agents. Does an agent $A$ (the taxi driver for example) have to treat an object $B$ (another vehicle) as an agent, or can it be treated merely as an object behaving according to the laws of physics, analogous to waves at the beach or leaves blowing in the wind? The key distinction is whether $B$'s behavior is best described as maximizing a performance measure whose value depends on agent $A$'s behavior.

*Single-agent*

*Multiagent*

For example, in chess, the opponent entity $B$ is trying to maximize its performance measure, which, by the rules of chess, minimizes agent $A$'s performance measure. Thus, chess is a **competitive** multiagent environment. On the other hand, in the taxi-driving environment, avoiding collisions maximizes the performance measure of all agents, so it is a partially

**cooperative** multiagent environment. It is also partially competitive because, for example, only one car can occupy a parking space.

*Competitive*

*Cooperative*

The agent-design problems in multiagent environments are often quite different from those in single-agent environments; for example, communication often emerges as a rational behavior in multiagent environments; in some competitive environments, randomized behavior is rational because it avoids the pitfalls of predictability.

**Deterministic** vs. **nondeterministic**. If the next state of the environment is completely determined by the current state and the action executed by the agent(s), then we say the environment is deterministic; otherwise, it is nondeterministic. In principle, an agent need not worry about uncertainty in a fully observable, deterministic environment. If the environment is partially observable, however, then it could *appear* to be nondeterministic.

*Deterministic*

*Nondeterministic*

Most real situations are so complex that it is impossible to keep track of all the unobserved aspects; for practical purposes, they must be treated as nondeterministic. Taxi driving is clearly nondeterministic in this sense, because one can never predict the behavior of traffic exactly; moreover, one's tires may blow out unexpectedly and one's engine may seize up without warning. The vacuum world as we described it is deterministic, but variations can include nondeterministic elements such as randomly appearing dirt and an unreliable suction mechanism (Exercise 2.VFIN).

One final note: the word **stochastic** is used by some as a synonym for "nondeterministic," but we make a distinction between the two terms; we say that a model of the environment is stochastic if it explicitly deals with probabilities (e.g., "there's a 25% chance of rain tomorrow") and "nondeterministic" if the possibilities are listed without being quantified (e.g., "there's a chance of rain tomorrow").

*Stochastic*

**EPISODIC VS. SEQUENTIAL:** In an episodic task environment, the agent's experience is divided into atomic episodes. In each episode the agent receives a percept and then performs a single action. Crucially, the next episode does not depend on the actions taken in previous episodes. Many classification tasks are episodic. For example, an agent that has to spot defective parts on an assembly line bases each decision on the current part, regardless of previous decisions; moreover, the current decision doesn't affect whether the next part is defective. In sequential environments, on the other hand, the current decision could affect all future decisions.[4] Chess and taxi driving are sequential: in both cases, short-term actions can have long-term consequences. Episodic environments are much simpler than sequential environments because the agent does not need to think ahead.

4 The word "sequential" is also used in computer science as the antonym of "parallel." The two meanings are largely unrelated.

*Episodic*

*Sequential*

*Static*

*Dynamic*

**STATIC VS. DYNAMIC:** If the environment can change while an agent is deliberating, then we say the environment is dynamic for that agent; otherwise, it is static. Static environments are easy to deal with because the agent need not keep looking at the world while it is deciding on an action, nor need it worry about the passage of time. Dynamic environments, on the other hand, are continuously asking the agent what it wants to do; if it hasn't decided yet, that counts as deciding to do nothing. If the environment itself does not change with the passage of time but the agent's performance score does, then we say the environment is **semidynamic**. Taxi driving is clearly dynamic: the other cars and the taxi itself keep moving while the driving algorithm dithers about what to do next. Chess, when played with a clock, is semidynamic. Crossword puzzles are static.

*Semidynamic*

**DISCRETE VS. CONTINUOUS:** The discrete/continuous distinction applies to the *state* of the environment, to the way *time* is handled, and to the *percepts* and *actions* of the agent. For example, the chess environment has a finite number of distinct states (excluding the clock). Chess also has a discrete set of percepts and actions. Taxi driving is a continuous-state and continuous-time problem: the speed and location of the taxi and of the other vehicles sweep through a range of continuous values and do so smoothly over time. Taxi-driving actions are also continuous (steering angles, etc.). Input from digital cameras is discrete, strictly speaking, but is typically treated as representing continuously varying intensities and locations.

*Discrete*

*Continuous*

**KNOWN VS. UNKNOWN:** Strictly speaking, this distinction refers not to the environment itself but to the agent's (or designer's) state of knowledge about the "laws of physics" of the environment. In a known environment, the outcomes (or outcome probabilities if the environment is nondeterministic) for all actions are given. Obviously, if the environment is unknown, the agent will have to learn how it works in order to make good decisions.

*Known*

*Unknown*

The distinction between known and unknown environments is not the same as the one between fully and partially observable environments. It is quite possible for a *known* environment to be *partially* observable—for example, in solitaire card games, I know the rules but am still unable to see the cards that have not yet been turned over. Conversely, an *unknown* environment can be *fully* observable—in a new video game, the screen may show the entire game state but I still don't know what the buttons do until I try them.

As noted on page 39, the performance measure itself may be unknown, either because the designer is not sure how to write it down correctly or because the ultimate user—whose preferences matter—is not known. For example, a taxi driver usually won't know whether a new passenger prefers a leisurely or speedy journey, a cautious or aggressive driving style. A virtual personal assistant starts out knowing nothing about the personal preferences of its new owner. In such cases, the agent may learn more about the performance measure based on further interactions with the designer or user. This, in turn, suggests that the task environment is necessarily viewed as a multiagent environment.

The hardest case is *partially observable, multiagent, nondeterministic, sequential, dynamic, continuous*, and *unknown*. Taxi driving is hard in all these senses, except that the driver's environment is mostly known. Driving a rented car in a new country with unfamiliar geography, different traffic laws, and nervous passengers is a lot more exciting.

Figure 2.6 lists the properties of a number of familiar environments. Note that the properties are not always cut and dried. For example, we have listed the medical-diagnosis task as single-agent because the disease process in a patient is not profitably modeled as an agent; but a medical-diagnosis system might also have to deal with recalcitrant patients and skeptical staff, so the environment could have a multiagent aspect. Furthermore, medical diagnosis is episodic if one conceives of the task as selecting a diagnosis given a list of symptoms; the problem is sequential if the task can include proposing a series of tests, evaluating progress over the course of treatment, handling multiple patients, and so on.

Figure 2.6

| Task Environment | Observable | Agents | Deterministic | Episodic | Static | Discrete |
|---|---|---|---|---|---|---|
| Crossword puzzle | Fully | Single | Deterministic | Sequential | Static | Discrete |
| Chess with a clock | Fully | Multi | Deterministic | Sequential | Semi | Discrete |
| Poker | Partially | Multi | Stochastic | Sequential | Static | Discrete |
| Backgammon | Fully | Multi | Stochastic | Sequential | Static | Discrete |
| Taxi driving | Partially | Multi | Stochastic | Sequential | Dynamic | Continuous |
| Medical diagnosis | Partially | Single | Stochastic | Sequential | Dynamic | Continuous |
| Image analysis | Fully | Single | Deterministic | Episodic | Semi | Continuous |
| Part-picking robot | Partially | Single | Stochastic | Episodic | Dynamic | Continuous |
| Refinery controller | Partially | Single | Stochastic | Sequential | Dynamic | Continuous |
| English tutor | Partially | Multi | Stochastic | Sequential | Dynamic | Discrete |

Examples of task environments and their characteristics.

We have not included a "known/unknown" column because, as explained earlier, this is not strictly a property of the environment. For some environments, such as chess and poker, it is quite easy to supply the agent with full knowledge of the rules, but it is nonetheless interesting to consider how an agent might learn to play these games without such knowledge.

The code repository associated with this book (aima.cs.berkeley.edu) includes multiple environment implementations, together with a general-purpose environment simulator for evaluating an agent's performance. Experiments are often carried out not for a single environment but for many environments drawn from an **environment class**. For example, to evaluate a taxi driver in simulated traffic, we would want to run many simulations with different traffic, lighting, and weather conditions. We are then interested in the agent's average performance over the environment class.

*Environment class*

## 2.4 The Structure of Agents

So far we have talked about agents by describing *behavior*—the action that is performed after any given sequence of percepts. Now we must bite the bullet and talk about how the insides work. The job of AI is to design an **agent program** that implements the agent function—the mapping from percepts to actions. We assume this program will run on some sort of computing device with physical sensors and actuators—we call this the **agent architecture**:

$$agent = architecture + program\ .$$

*Agent program*

*Agent architecture*

Obviously, the program we choose has to be one that is appropriate for the architecture. If the program is going to recommend actions like *Walk*, the architecture had better have legs. The architecture might be just an ordinary PC, or it might be a robotic car with several onboard computers, cameras, and other sensors. In general, the architecture makes the percepts from the sensors available to the program, runs the program, and feeds the program's action choices to the actuators as they are generated. Most of this book is about designing agent programs, although Chapters 25 and 26 deal directly with the sensors and actuators.

### 2.4.1 Agent programs

The agent programs that we design in this book all have the same skeleton: they take the current percept as input from the sensors and return an action to the actuators.[5] Notice the difference between the agent program, which takes the current percept as input, and the agent function, which may depend on the entire percept history. The agent program has no choice but to take just the current percept as input because nothing more is available from the environment; if the agent's actions need to depend on the entire percept sequence, the agent will have to remember the percepts.

5 There are other choices for the agent program skeleton; for example, we could have the agent programs be **coroutines** that run asynchronously with the environment. Each such coroutine has an input and output port and consists of a loop that reads the input port for percepts and writes actions to the output port.

We describe the agent programs in the simple pseudocode language that is defined in Appendix B. (The online code repository contains implementations in real programming languages.) For example, Figure 2.7 shows a rather trivial agent program that keeps track of the percept sequence and then uses it to index into a table of actions to decide what to do. The table—an example of which is given for the vacuum world in Figure 2.3—represents explicitly the agent function that the agent program embodies. To build a rational agent in this way, we as designers must construct a table that contains the appropriate action for every possible percept sequence.

Figure 2.7

```
function TABLE-DRIVEN-AGENT(percept) returns an action
   persistent: percepts, a sequence, initially empty
               table, a table of actions, indexed by percept sequences, initially fully specified

   append percept to the end of percepts
   action ← LOOKUP(percepts, table)
   return action
```

The TABLE-DRIVEN-AGENT program is invoked for each new percept and returns an action each time. It retains the complete percept sequence in memory.

It is instructive to consider why the table-driven approach to agent construction is doomed to failure. Let $P$ be the set of possible percepts and let $T$ be the lifetime of the agent (the total number of percepts it will receive). The lookup table will contain $\sum_{t=1}^{T} |P|^t$ entries. Consider the automated taxi: the visual input from a single camera (eight cameras is typical) comes in at the rate of roughly 70 megabytes per second (30 frames per second, $1080 \times 720$ pixels with 24 bits of color information). This gives a lookup table with over $10^{600,000,000,000}$ entries for an hour's driving. Even the lookup table for chess—a tiny, well-behaved fragment of the real world—has (it turns out) at least $10^{150}$ entries. In comparison, the number of atoms in the observable universe is less than $10^{80}$. The daunting size of these tables means that (a) no physical agent in this universe will have the space to store the table; (b) the designer would not have time to create the table; and (c) no agent could ever learn all the right table entries from its experience.

Despite all this, TABLE-DRIVEN-AGENT *does* do what we want, assuming the table is filled in correctly: it implements the desired agent function.

*The key challenge for AI is to find out how to write programs that, to the extent possible, produce rational behavior from a smallish program rather than from a vast table.*

We have many examples showing that this can be done successfully in other areas: for example, the huge tables of square roots used by engineers and schoolchildren prior to the 1970s have now been replaced by a five-line program for Newton's method running on electronic calculators. The question is, can AI do for general intelligent behavior what Newton did for square roots? We believe the answer is yes.

In the remainder of this section, we outline four basic kinds of agent programs that embody the principles underlying almost all intelligent systems:

- Simple reflex agents;
- Model-based reflex agents;
- Goal-based agents; and
- Utility-based agents.

Each kind of agent program combines particular components in particular ways to generate actions. Section 2.4.6 explains in general terms how to convert all these agents into *learning agents* that can improve the performance of their components so as to generate better actions. Finally, Section 2.4.7 describes the variety of ways in which the components themselves can be represented within the agent. This variety provides a major organizing principle for the field and for the book itself.

### 2.4.2 Simple reflex agents

The simplest kind of agent is the **simple reflex agent**. These agents select actions on the basis of the *current* percept, ignoring the rest of the percept history. For example, the vacuum agent whose agent function is tabulated in Figure 2.3 is a simple reflex agent, because its decision is based only on the current location and on whether that location contains dirt. An agent program for this agent is shown in Figure 2.8.

Figure 2.8

**function** REFLEX-VACUUM-AGENT([*location*,*status*]) **returns** an action
  **if** *status* = *Dirty* **then return** *Suck*
  **else if** *location* = *A* **then return** *Right*
  **else if** *location* = *B* **then return** *Left*

The agent program for a simple reflex agent in the two-location vacuum environment. This program implements the agent function tabulated in Figure 2.3.

*Simple reflex agent*

Notice that the vacuum agent program is very small indeed compared to the corresponding table. The most obvious reduction comes from ignoring the percept history, which cuts down the number of relevant percept sequences from $4^T$ to just 4. A further, small reduction comes from the fact that when the current square is dirty, the action does not depend on the location. Although we have written the agent program using if-then-else statements, it is simple enough that it can also be implemented as a Boolean circuit.

Simple reflex behaviors occur even in more complex environments. Imagine yourself as the driver of the automated taxi. If the car in front brakes and its brake lights come on, then you should notice this and initiate braking. In other words, some processing is done on the visual input to establish the condition we call “The car in front is braking.” Then, this triggers some established connection in the agent program to the action “initiate braking.” We call such a connection a **condition–action rule**,[6] written as

6 Also called **situation–action rules**, **productions**, or **if–then rules**.

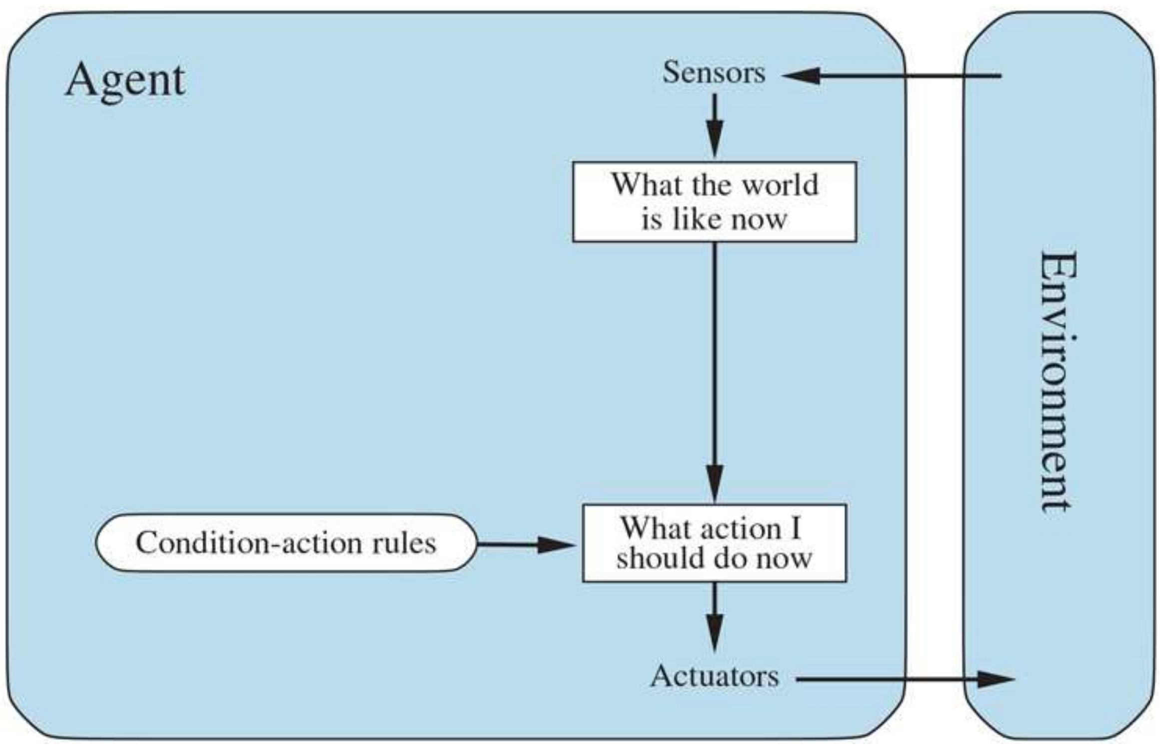
> **if** *car-in-front-is-braking* **then** *initiate-braking*.

*condition–action rule*

Humans also have many such connections, some of which are learned responses (as for driving) and some of which are innate reflexes (such as blinking when something approaches the eye). In the course of the book, we show several different ways in which such connections can be learned and implemented.

The program in Figure 2.8 is specific to one particular vacuum environment. A more general and flexible approach is first to build a general-purpose interpreter for condition–action rules and then to create rule sets for specific task environments. Figure 2.9 gives the structure of this general program in schematic form, showing how the condition–action rules allow the agent to make the connection from percept to action. Do not worry if this seems trivial; it gets more interesting shortly.

Figure 2.9




Schematic diagram of a simple reflex agent. We use rectangles to denote the current internal state of the agent's decision process, and ovals to represent the background information used in the process.

An agent program for Figure 2.9 is shown in Figure 2.10. The Interpret-Input function generates an abstracted description of the current state from the percept, and the Rule-Match function returns the first rule in the set of rules that matches the given state description. Note that the description in terms of “rules” and “matching” is purely conceptual; as noted above, actual implementations can be as simple as a collection of logic gates implementing a Boolean circuit. Alternatively, a “neural” circuit can be used, where the logic gates are replaced by the nonlinear units of artificial neural networks (see Chapter 21).

Figure 2.10

**function** Simple-Reflex-Agent(*percept*) **returns** an action
  **persistent**: *rules*, a set of condition–action rules

  *state* ← Interpret-Input(*percept*)
  *rule* ← Rule-Match(*state*, *rules*)
  *action* ← *rule*.Action
  **return** *action*

A simple reflex agent. It acts according to a rule whose condition matches the current state, as defined by the percept.

Simple reflex agents have the admirable property of being simple, but they are of limited intelligence. The agent in Figure 2.10 will work *only if the correct decision can be made on the basis of just the current percept—that is, only if the environment is fully observable.*

Even a little bit of unobservability can cause serious trouble. For example, the braking rule given earlier assumes that the condition *car-in-front-is-braking* can be determined from the current percept—a single frame of video. This works if the car in front has a centrally mounted (and hence uniquely identifiable) brake light. Unfortunately, older models have different configurations of taillights, brake lights, and turn-signal lights, and it is not always possible to tell from a single image whether the car is braking or simply has its taillights on. A simple reflex agent driving behind such a car would either brake continuously and unnecessarily, or, worse, never brake at all.

We can see a similar problem arising in the vacuum world. Suppose that a simple reflex vacuum agent is deprived of its location sensor and has only a dirt sensor. Such an agent has just two possible percepts: $[Dirty]$ and $[Clean]$. It can *Suck* in response to $[Dirty]$; what should it do in response to $[Clean]$? Moving $Left$ fails (forever) if it happens to start in square $A$, and moving $Right$ fails (forever) if it happens to start in square $B$. Infinite loops are often unavoidable for simple reflex agents operating in partially observable environments.

Escape from infinite loops is possible if the agent can **randomize** its actions. For example, if the vacuum agent perceives $[Clean]$, it might flip a coin to choose between $Right$ and $Left$. It is easy to show that the agent will reach the other square in an average of two steps. Then, if that square is dirty, the agent will clean it and the task will be complete. Hence, a randomized simple reflex agent might outperform a deterministic simple reflex agent.

*Randomization*

We mentioned in Section 2.3 that randomized behavior of the right kind can be rational in some multiagent environments. In single-agent environments, randomization is usually *not* rational. It is a useful trick that helps a simple reflex agent in some situations, but in most cases we can do much better with more sophisticated deterministic agents.

## 2.4.3 Model-based reflex agents

The most effective way to handle partial observability is for the agent to *keep track of the part of the world it can't see now*. That is, the agent should maintain some sort of **internal state** that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state. For the braking problem, the internal state is not too extensive—just the previous frame from the camera, allowing the agent to detect when two red lights at the edge of the vehicle go on or off simultaneously. For other driving tasks such as changing lanes, the agent needs to keep track of where the other cars are if it can't see them all at once. And for any driving to be possible at all, the agent needs to keep track of where its keys are.

*Internal state*

Updating this internal state information as time goes by requires two kinds of knowledge to be encoded in the agent program in some form. First, we need some information about how the world changes over time, which can be divided roughly into two parts: the effects of the agent's actions and how the world evolves independently of the agent. For example, when the agent turns the steering wheel clockwise, the car turns to the right, and when it's raining the car's cameras can get wet. This knowledge about "how the world works"—whether implemented in simple Boolean circuits or in complete scientific theories—is called a **transition model** of the world.

*Transition model*

Second, we need some information about how the state of the world is reflected in the agent's percepts. For example, when the car in front initiates braking, one or more illuminated red regions appear in the forward-facing camera image, and, when the camera gets wet, droplet-shaped objects appear in the image partially obscuring the road. This kind of knowledge is called a **sensor model**.

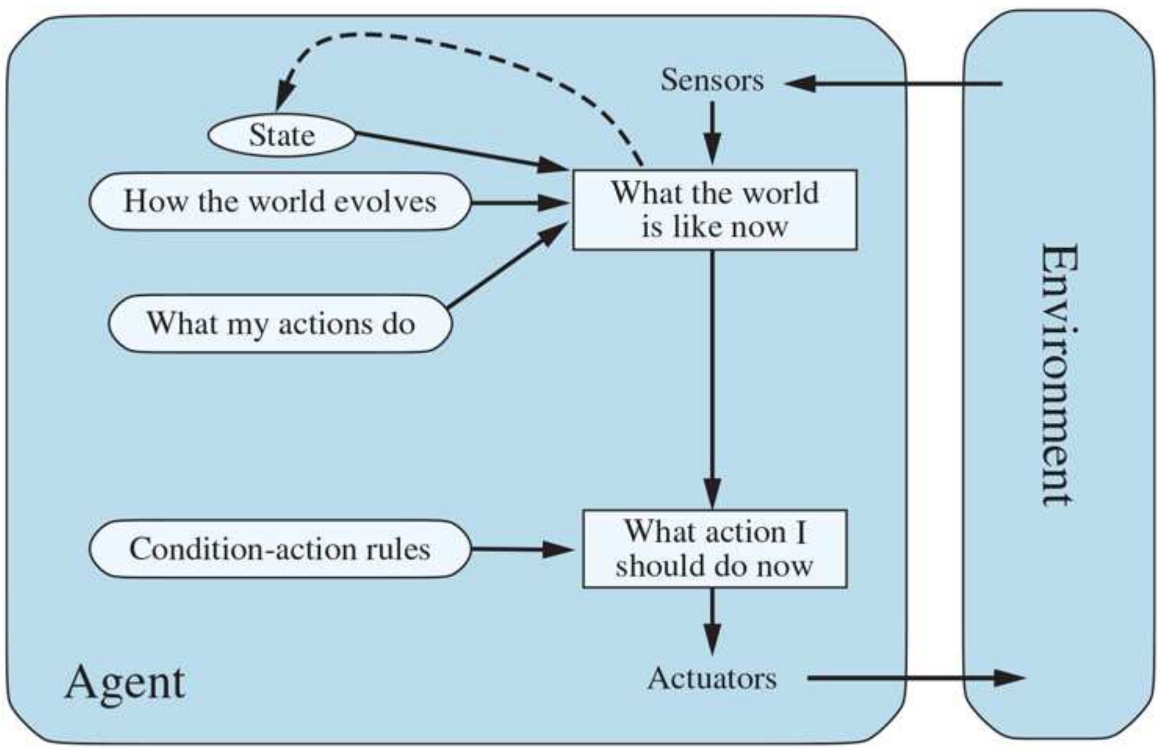
*Sensor model*

Together, the transition model and sensor model allow an agent to keep track of the state of the world—to the extent possible given the limitations of the agent's sensors. An agent that uses such models is called a **model-based agent**.

*Model-based agent*

Figure 2.11 gives the structure of the model-based reflex agent with internal state, showing how the current percept is combined with the old internal state to generate the updated description of the current state, based on the agent's model of how the world works. The agent program is shown in Figure 2.12. The interesting part is the function UPDATE-STATE, which is responsible for creating the new internal state description. The details of how models and states are represented vary widely depending on the type of environment and the particular technology used in the agent design.

Figure 2.11




A model-based reflex agent.

Figure 2.12

**function** MODEL-BASED-REFLEX-AGENT(*percept*) **returns** an action
  **persistent**: *state*, the agent's current conception of the world state
    *transition_model*, a description of how the next state depends on
      the current state and action
    *sensor_model*, a description of how the current world state is reflected
      in the agent's percepts
    *rules*, a set of condition–action rules
    *action*, the most recent action, initially none

  *state* ← UPDATE-STATE(*state*, *action*, *percept*, *transition_model*, *sensor_model*)
  *rule* ← RULE-MATCH(*state*, *rules*)
  *action* ← *rule*.ACTION
  **return** *action*

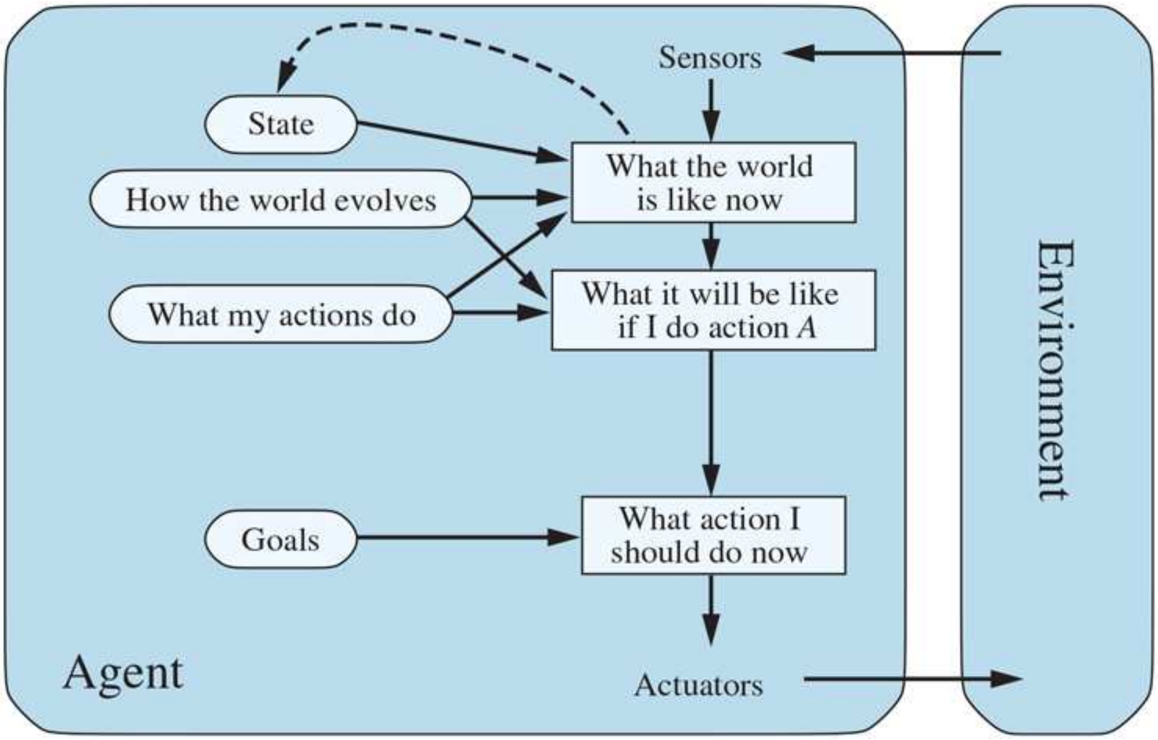
A model-based reflex agent. It keeps track of the current state of the world, using an internal model. It then chooses an action in the same way as the reflex agent.

Regardless of the kind of representation used, it is seldom possible for the agent to determine the current state of a partially observable environment *exactly*. Instead, the box labeled "what the world is like now" (Figure 2.11) represents the agent's "best guess" (or sometimes best guesses, if the agent entertains multiple possibilities). For example, an automated taxi may not be able to see around the large truck that has stopped in front of it and can only guess about what may be causing the hold-up. Thus, uncertainty about the current state may be unavoidable, but the agent still has to make a decision.

## 2.4.4 Goal-based agents

Knowing something about the current state of the environment is not always enough to decide what to do. For example, at a road junction, the taxi can turn left, turn right, or go straight on. The correct decision depends on where the taxi is trying to get to. In other words, as well as a current state description, the agent needs some sort of **goal** information that describes situations that are desirable—for example, being at a particular destination. The agent program can combine this with the model (the same information as was used in the model-based reflex agent) to choose actions that achieve the goal. Figure 2.13 shows the goal-based agent's structure.

Figure 2.13




A model-based, goal-based agent. It keeps track of the world state as well as a set of goals it is trying to achieve, and chooses an action that will (eventually) lead to the achievement of its goals.

*Goal*

Sometimes goal-based action selection is straightforward—for example, when goal satisfaction results immediately from a single action. Sometimes it will be more tricky—for example, when the agent has to consider long sequences of twists and turns in order to find a way to achieve the goal. **Search** (Chapters 3 to 5) and **planning** (Chapter 11) are the subfields of AI devoted to finding action sequences that achieve the agent's goals.

Notice that decision making of this kind is fundamentally different from the condition–action rules described earlier, in that it involves consideration of the future—both "What will happen if I do such-and-such?" and "Will that make me happy?" In the reflex agent designs, this information is not explicitly represented, because the built-in rules map directly from percepts to actions. The reflex agent brakes when it sees brake lights, period. It has no idea why. A goal-based agent brakes when it sees brake lights because that's the only action that it predicts will achieve its goal of not hitting other cars.

Although the goal-based agent appears less efficient, it is more flexible because the knowledge that supports its decisions is represented explicitly and can be modified. For example, a goal-based agent's behavior can easily be changed to go to a different destination, simply by specifying that destination as the goal. The reflex agent's rules for when to turn and when to go straight will work only for a single destination; they must all be replaced to go somewhere new.

### 2.4.5 Utility-based agents

Goals alone are not enough to generate high-quality behavior in most environments. For example, many action sequences will get the taxi to its destination (thereby achieving the goal), but some are quicker, safer, more reliable, or cheaper than others. Goals just provide a crude binary distinction between "happy" and "unhappy" states. A more general performance measure should allow a comparison of different world states according to exactly how happy they would make the agent. Because "happy" does not sound very scientific, economists and computer scientists use the term **utility** instead.[7]

7 The word "utility" here refers to "the quality of being useful," not to the electric company or waterworks.

*Utility*

We have already seen that a performance measure assigns a score to any given sequence of environment states, so it can easily distinguish between more and less desirable ways of getting to the taxi's destination. An agent's **utility function** is essentially an internalization of the performance measure. Provided that the internal utility function and the external performance measure are in agreement, an agent that chooses actions to maximize its utility will be rational according to the external performance measure.

*Utility function*

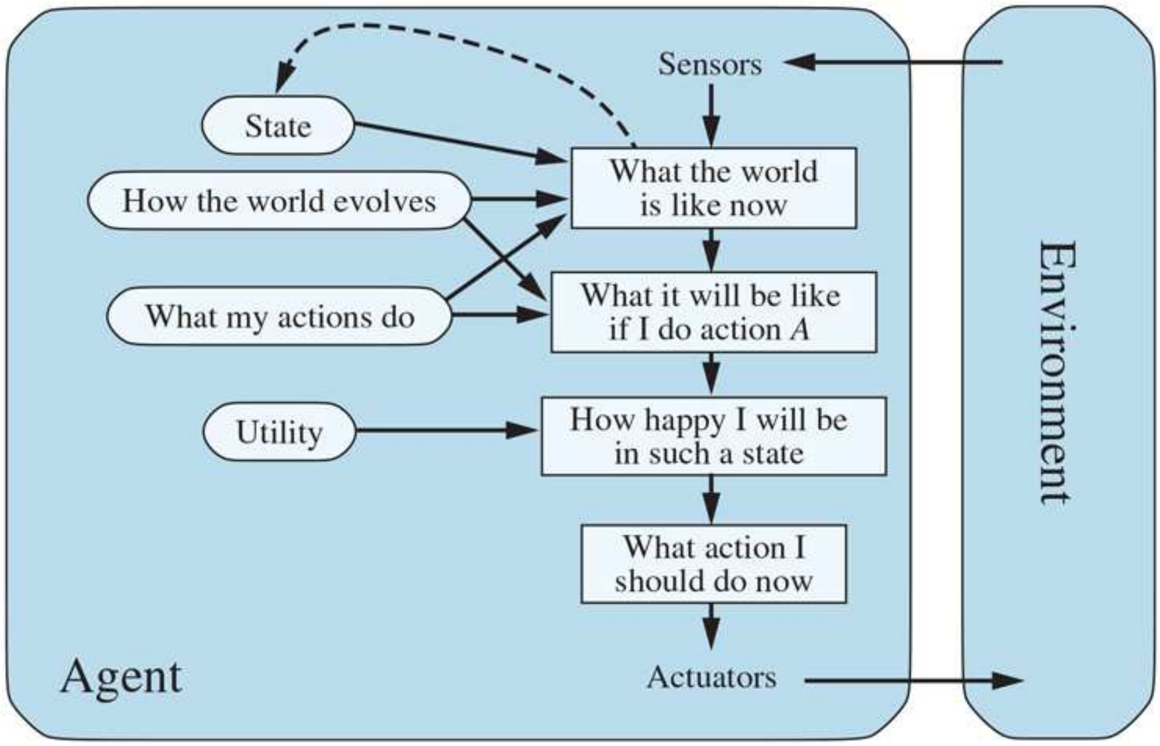
Let us emphasize again that this is not the *only* way to be rational—we have already seen a rational agent program for the vacuum world (Figure 2.8) that has no idea what its utility function is—but, like goal-based agents, a utility-based agent has many advantages in terms of flexibility and learning. Furthermore, in two kinds of cases, goals are inadequate but a utility-based agent can still make rational decisions. First, when there are conflicting goals, only some of which can be achieved (for example, speed and safety), the utility function specifies the appropriate tradeoff. Second, when there are several goals that the agent can aim for, none of which can be achieved with certainty, utility provides a way in which the likelihood of success can be weighed against the importance of the goals.

Partial observability and nondeterminism are ubiquitous in the real world, and so, therefore, is decision making under uncertainty. Technically speaking, a rational utility-based agent chooses the action that maximizes the **expected utility** of the action outcomes—that is, the utility the agent expects to derive, on average, given the probabilities and utilities of each outcome. (Appendix A defines expectation more precisely.) In Chapter 16, we show that any rational agent must behave *as if* it possesses a utility function whose expected value it tries to maximize. An agent that possesses an *explicit* utility function can make rational decisions with a general-purpose algorithm that does not depend on the specific utility function being maximized. In this way, the "global" definition of rationality—designating as rational those agent functions that have the highest performance—is turned into a "local" constraint on rational-agent designs that can be expressed in a simple program.

*Expected utility*

The utility-based agent structure appears in Figure 2.14. Utility-based agent programs appear in Chapters 16 and 17, where we design decision-making agents that must handle the uncertainty inherent in nondeterministic or partially observable environments. Decision making in multiagent environments is also studied in the framework of utility theory, as explained in Chapter 18.

Figure 2.14




A model-based, utility-based agent. It uses a model of the world, along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome states, weighted by the probability of the outcome.

At this point, the reader may be wondering, "Is it that simple? We just build agents that maximize expected utility, and we're done?" It's true that such agents would be intelligent, but it's not simple. A utility-based agent has to model and keep track of its environment, tasks that have involved a great deal of research on perception, representation, reasoning, and learning. The results of this research fill many of the chapters of this book. Choosing the utility-maximizing course of action is also a difficult task, requiring ingenious algorithms that fill several more chapters. Even with these algorithms, perfect rationality is usually unachievable in practice because of computational complexity, as we noted in Chapter 1. We also note that not all utility-based agents are model-based; we will see in Chapters 22 and 26 that a **model-free agent** can learn what action is best in a particular situation without ever learning exactly how that action changes the environment.

*Model-free agent*

Finally, all of this assumes that the designer can specify the utility function correctly; Chapters 17, 18, and 22 consider the issue of unknown utility functions in more depth.

### 2.4.6 Learning agents

We have described agent programs with various methods for selecting actions. We have not, so far, explained how the agent programs *come into being*. In his famous early paper, Turing (1950) considers the idea of actually programming his intelligent machines by hand. He estimates how much work this might take and concludes, “Some more expeditious method seems desirable.” The method he proposes is to build learning machines and then to teach them. In many areas of AI, this is now the preferred method for creating state-of-the-art systems. Any type of agent (model-based, goal-based, utility-based, etc.) can be built as a learning agent (or not).

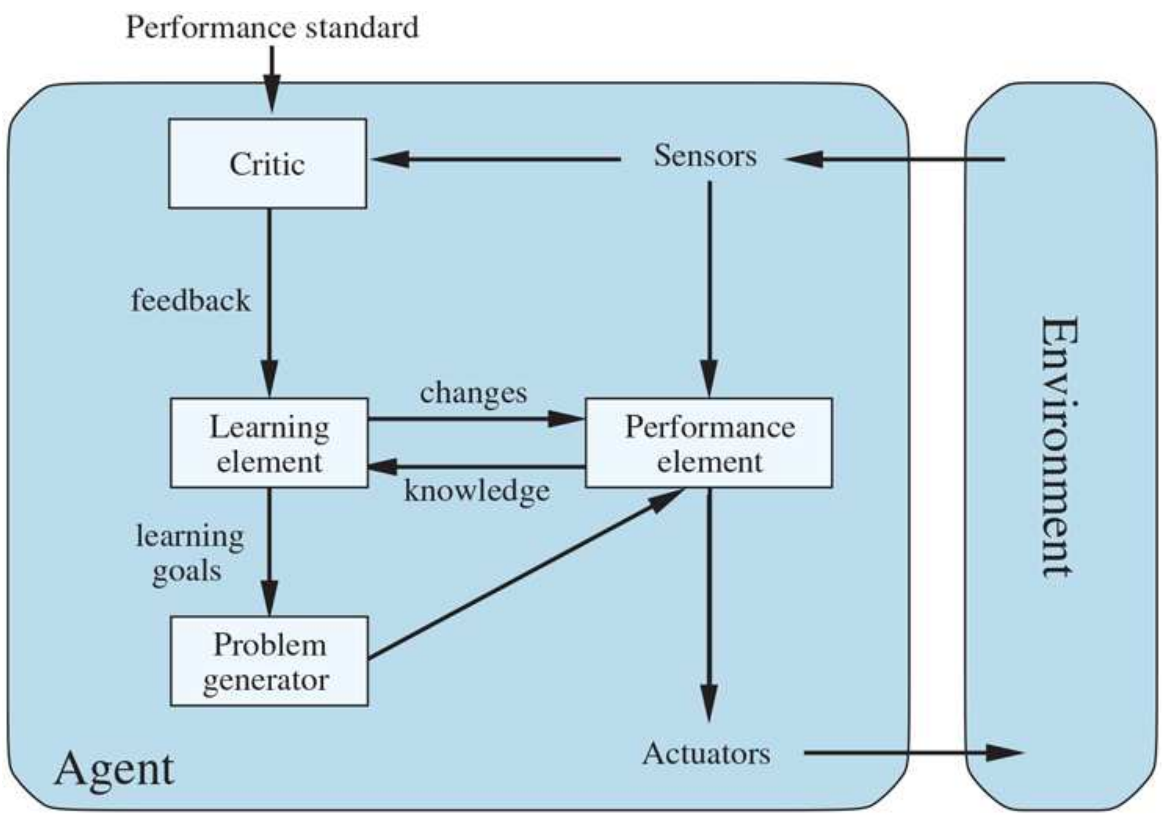
Learning has another advantage, as we noted earlier: it allows the agent to operate in initially unknown environments and to become more competent than its initial knowledge alone might allow. In this section, we briefly introduce the main ideas of learning agents. Throughout the book, we comment on opportunities and methods for learning in particular kinds of agents. Chapters 19–22 go into much more depth on the learning algorithms themselves.

*Learning element*

*Performance element*

A learning agent can be divided into four conceptual components, as shown in Figure 2.15. The most important distinction is between the **learning element**, which is responsible for making improvements, and the **performance element**, which is responsible for selecting external actions. The performance element is what we have previously considered to be the entire agent: it takes in percepts and decides on actions. The learning element uses feedback from the **critic** on how the agent is doing and determines how the performance element should be modified to do better in the future.

Figure 2.15




A general learning agent. The "performance element" box represents what we have previously considered to be the whole agent program. Now, the "learning element" box gets to modify that program to improve its performance.

*Critic*

The design of the learning element depends very much on the design of the performance element. When trying to design an agent that learns a certain capability, the first question is not "How am I going to get it to learn this?" but "What kind of performance element will my agent use to do this once it has learned how?” Given a design for the performance element, learning mechanisms can be constructed to improve every part of the agent.

The critic tells the learning element how well the agent is doing with respect to a fixed performance standard. The critic is necessary because the percepts themselves provide no indication of the agent’s success. For example, a chess program could receive a percept indicating that it has checkmated its opponent, but it needs a performance standard to know that this is a good thing; the percept itself does not say so. It is important that the performance standard be fixed. Conceptually, one should think of it as being outside the agent altogether because the agent must not modify it to fit its own behavior.

The last component of the learning agent is the **problem generator**. It is responsible for suggesting actions that will lead to new and informative experiences. If the performance element had its way, it would keep doing the actions that are best, given what it knows, but if the agent is willing to explore a little and do some perhaps suboptimal actions in the short run, it might discover much better actions for the long run. The problem generator’s job is to suggest these exploratory actions. This is what scientists do when they carry out experiments. Galileo did not think that dropping rocks from the top of a tower in Pisa was valuable in itself. He was not trying to break the rocks or to modify the brains of unfortunate pedestrians. His aim was to modify his own brain by identifying a better theory of the motion of objects.

*Problem generator*

The learning element can make changes to any of the “knowledge” components shown in the agent diagrams (Figures 2.9, 2.11, 2.13, and 2.14). The simplest cases involve learning directly from the percept sequence. Observation of pairs of successive states of the environment can allow the agent to learn “What my actions do” and “How the world evolves” in response to its actions. For example, if the automated taxi exerts a certain braking pressure when driving on a wet road, then it will soon find out how much deceleration is actually achieved, and whether it skids off the road. The problem generator might identify certain parts of the model that are in need of improvement and suggest experiments, such as trying out the brakes on different road surfaces under different conditions.

Improving the model components of a model-based agent so that they conform better with reality is almost always a good idea, regardless of the external performance standard. (In some cases, it is better from a computational point of view to have a simple but slightly inaccurate model rather than a perfect but fiendishly complex model.) Information from the external standard is needed when trying to learn a reflex component or a utility function.

For example, suppose the taxi-driving agent receives no tips from passengers who have been thoroughly shaken up during the trip. The external performance standard must inform the agent that the loss of tips is a negative contribution to its overall performance; then the agent might be able to learn that violent maneuvers do not contribute to its own utility. In a sense, the performance standard distinguishes part of the incoming percept as a **reward** (or **penalty**) that provides direct feedback on the quality of the agent's behavior. Hard-wired performance standards such as pain and hunger in animals can be understood in this way.

---

*Reward*

---

*Penalty*

More generally, *human choices* can provide information about human preferences. For example, suppose the taxi does not know that people generally don't like loud noises, and settles on the idea of blowing its horn continuously as a way of ensuring that pedestrians know it's coming. The consequent human behavior—covering ears, using bad language, and possibly cutting the wires to the horn—would provide evidence to the agent with which to update its utility function. This issue is discussed further in Chapter 22.

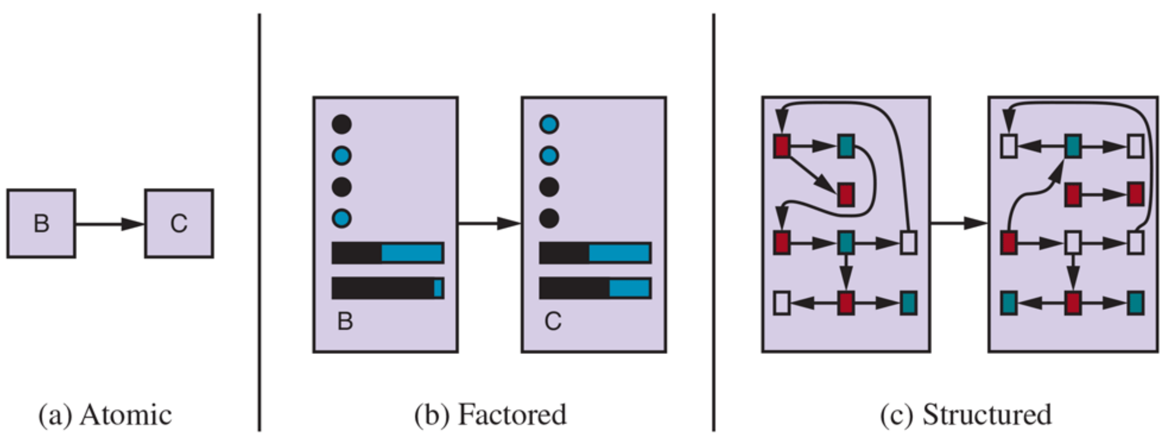
In summary, agents have a variety of components, and those components can be represented in many ways within the agent program, so there appears to be great variety among learning methods. There is, however, a single unifying theme. Learning in intelligent agents can be summarized as a process of modification of each component of the agent to bring the components into closer agreement with the available feedback information, thereby improving the overall performance of the agent.

### 2.4.7 How the components of agent programs work

We have described agent programs (in very high-level terms) as consisting of various components, whose function it is to answer questions such as: “What is the world like now?” “What action should I do now?” “What do my actions do?” The next question for a student of AI is, “How on Earth do these components work?” It takes about a thousand pages to begin to answer that question properly, but here we want to draw the reader’s attention to some basic distinctions among the various ways that the components can represent the environment that the agent inhabits.

Roughly speaking, we can place the representations along an axis of increasing complexity and expressive power—atomic, factored, and structured. To illustrate these ideas, it helps to consider a particular agent component, such as the one that deals with “What my actions do.” This component describes the changes that might occur in the environment as the result of taking an action, and Figure 2.16 provides schematic depictions of how those transitions might be represented.

Figure 2.16




Three ways to represent states and the transitions between them. (a) Atomic representation: a state (such as B or C) is a black box with no internal structure; (b) Factored representation: a state consists of a vector of attribute values; values can be Boolean, real-valued, or one of a fixed set of symbols. (c) Structured representation: a state includes objects, each of which may have attributes of its own as well as relationships to other objects.

In an **atomic representation** each state of the world is indivisible—it has no internal structure. Consider the task of finding a driving route from one end of a country to the other via some sequence of cities (we address this problem in Figure 3.1 on page 64). For the purposes of solving this problem, it may suffice to reduce the state of the world to just the name of the city we are in—a single atom of knowledge, a "black box" whose only discernible property is that of being identical to or different from another black box. The standard algorithms underlying search and game-playing (Chapters 3–5), hidden Markov models (Chapter 14), and Markov decision processes (Chapter 17) all work with atomic representations.

*Atomic representation*

A **factored representation** splits up each state into a fixed set of **variables** or **attributes**, each of which can have a **value**. Consider a higher-fidelity description for the same driving problem, where we need to be concerned with more than just atomic location in one city or another; we might need to pay attention to how much gas is in the tank, our current GPS coordinates, whether or not the oil warning light is working, how much money we have for tolls, what station is on the radio, and so on. While two different atomic states have nothing in common—they are just different black boxes—two different factored states can share some attributes (such as being at some particular GPS location) and not others (such as having lots of gas or having no gas); this makes it much easier to work out how to turn one state into another. Many important areas of AI are based on factored representations, including constraint satisfaction algorithms (Chapter 6), propositional logic (Chapter 7), planning (Chapter 11), Bayesian networks (Chapters 12–16), and various machine learning algorithms.

*Factored representation*

*Variable*

*Attribute*

*Value*

For many purposes, we need to understand the world as having *things* in it that are *related* to each other, not just variables with values. For example, we might notice that a large truck ahead of us is reversing into the driveway of a dairy farm, but a loose cow is blocking the truck's path. A factored representation is unlikely to be pre-equipped with the attribute *TruckAheadBackingIntoDairyFarmDrivewayBlockedByLooseCow* with value *true* or *false*. Instead, we would need a **structured representation**, in which objects such as cows and trucks and their various and varying relationships can be described explicitly (see Figure 2.16(c)). Structured representations underlie relational databases and first-order logic (Chapters 8, 9, and 10), first-order probability models (Chapter 15), and much of natural language understanding (Chapters 23 and 24). In fact, much of what humans express in natural language concerns objects and their relationships.

*Structured representation*

As we mentioned earlier, the axis along which atomic, factored, and structured representations lie is the axis of increasing **expressiveness**. Roughly speaking, a more expressive representation can capture, at least as concisely, everything a less expressive one can capture, plus some more. Often, the more expressive language is *much* more concise; for example, the rules of chess can be written in a page or two of a structured-representation language such as first-order logic but require thousands of pages when written in a factored-representation language such as propositional logic and around $10^{38}$ pages when written in an atomic language such as that of finite-state automata. On the other hand, reasoning and learning become more complex as the expressive power of the representation increases. To gain the benefits of expressive representations while avoiding their drawbacks, intelligent systems for the real world may need to operate at all points along the axis simultaneously.

*Expressiveness*

*Localist representation*

Another axis for representation involves the mapping of concepts to locations in physical memory, whether in a computer or in a brain. If there is a one-to-one mapping between concepts and memory locations, we call that a **localist representation**. On the other hand, if the representation of a concept is spread over many memory locations, and each memory location is employed as part of the representation of multiple different concepts, we call that a **distributed representation**. Distributed representations are more robust against noise and information loss. With a localist representation, the mapping from concept to memory location is arbitrary, and if a transmission error garbles a few bits, we might confuse *Truck* with the unrelated concept *Truce*. But with a distributed representation, you can think of each concept representing a point in multidimensional space, and if you garble a few bits you move to a nearby point in that space, which will have similar meaning.

*Distributed representation*

## Summary

This chapter has been something of a whirlwind tour of AI, which we have conceived of as the science of agent design. The major points to recall are as follows:

- An **agent** is something that perceives and acts in an environment. The **agent function** for an agent specifies the action taken by the agent in response to any percept sequence.
- The **performance measure** evaluates the behavior of the agent in an environment. A **rational agent** acts so as to maximize the expected value of the performance measure, given the percept sequence it has seen so far.
- A **task environment** specification includes the performance measure, the external environment, the actuators, and the sensors. In designing an agent, the first step must always be to specify the task environment as fully as possible.
- Task environments vary along several significant dimensions. They can be fully or partially observable, single-agent or multiagent, deterministic or nondeterministic, episodic or sequential, static or dynamic, discrete or continuous, and known or unknown.
- In cases where the performance measure is unknown or hard to specify correctly, there is a significant risk of the agent optimizing the wrong objective. In such cases the agent design should reflect uncertainty about the true objective.
- The **agent program** implements the agent function. There exists a variety of basic agent program designs reflecting the kind of information made explicit and used in the decision process. The designs vary in efficiency, compactness, and flexibility. The appropriate design of the agent program depends on the nature of the environment.
- **Simple reflex agents** respond directly to percepts, whereas **model-based reflex agents** maintain internal state to track aspects of the world that are not evident in the current percept. **Goal-based agents** act to achieve their goals, and **utility-based agents** try to maximize their own expected “happiness.”
- All agents can improve their performance through **learning**.

## Bibliographical and Historical Notes

The central role of action in intelligence—the notion of practical reasoning—goes back at least as far as Aristotle's *Nicomachean Ethics*. Practical reasoning was also the subject of McCarthy's influential paper "Programs with Common Sense" (1958). The fields of robotics and control theory are, by their very nature, concerned principally with physical agents. The concept of a **controller** in control theory is identical to that of an agent in AI. Perhaps surprisingly, AI has concentrated for most of its history on isolated components of agents—question-answering systems, theorem-provers, vision systems, and so on—rather than on whole agents. The discussion of agents in the text by Genesereth and Nilsson (1987) was an influential exception. The whole-agent view is now widely accepted and is a central theme in recent texts (Padgham and Winikoff, 2004; Jones, 2007; Poole and Mackworth, 2017).

---

*Controller*

Chapter 1 traced the roots of the concept of rationality in philosophy and economics. In AI, the concept was of peripheral interest until the mid-1980s, when it began to suffuse many discussions about the proper technical foundations of the field. A paper by Jon Doyle (1983) predicted that rational agent design would come to be seen as the core mission of AI, while other popular topics would spin off to form new disciplines.

Careful attention to the properties of the environment and their consequences for rational agent design is most apparent in the control theory tradition—for example, classical control systems (Dorf and Bishop, 2004; Kirk, 2004) handle fully observable, deterministic environments; stochastic optimal control (Kumar and Varaiya, 1986; Bertsekas and Shreve, 2007) handles partially observable, stochastic environments; and hybrid control (Henzinger and Sastry, 1998; Cassandras and Lygeros, 2006) deals with environments containing both discrete and continuous elements. The distinction between fully and partially observable environments is also central in the **dynamic programming** literature developed in the field of operations research (Puterman, 1994), which we discuss in Chapter 17.

Although simple reflex agents were central to behaviorist psychology (see Chapter 1), most AI researchers view them as too simple to provide much leverage. (Rosenschein (1985) and Brooks (1986) questioned this assumption; see Chapter 26.) A great deal of work has gone into finding efficient algorithms for keeping track of complex environments (Bar-Shalom *et al.*, 2001; Choset *et al.*, 2005; Simon, 2006), most of it in the probabilistic setting.

Goal-based agents are presupposed in everything from Aristotle's view of practical reasoning to McCarthy's early papers on logical AI. Shakey the Robot (Fikes and Nilsson, 1971; Nilsson, 1984) was the first robotic embodiment of a logical, goal-based agent. A full logical analysis of goal-based agents appeared in Genesereth and Nilsson (1987), and a goal-based programming methodology called agent-oriented programming was developed by Shoham (1993). The agent-based approach is now extremely popular in software engineering (Ciancarini and Wooldridge, 2001). It has also infiltrated the area of operating systems, where **autonomic computing** refers to computer systems and networks that monitor and control themselves with a perceive–act loop and machine learning methods (Kephart and Chess, 2003). Noting that a collection of agent programs designed to work well together in a true multiagent environment necessarily exhibits modularity—the programs share no internal state and communicate with each other only through the environment—it is common within the field of **multiagent systems** to design the agent program of a single agent as a collection of autonomous sub-agents. In some cases, one can even prove that the resulting system gives the same optimal solutions as a monolithic design.

---

*Autonomic computing*

The goal-based view of agents also dominates the cognitive psychology tradition in the area of problem solving, beginning with the enormously influential *Human Problem Solving* (Newell and Simon, 1972) and running through all of Newell's later work (Newell, 1990). Goals, further analyzed as *desires* (general) and *intentions* (currently pursued), are central to the influential theory of agents developed by Michael Bratman (1987).

As noted in Chapter 1, the development of utility theory as a basis for rational behavior goes back hundreds of years. In AI, early research eschewed utilities in favor of goals, with some exceptions (Feldman and Sproull, 1977). The resurgence of interest in probabilistic methods in the 1980s led to the acceptance of maximization of expected utility as the most general framework for decision making (Horvitz *et al.*, 1988). The text by Pearl (1988) was the first in AI to cover probability and utility theory in depth; its exposition of practical methods for reasoning and decision making under uncertainty was probably the single biggest factor in the rapid shift towards utility-based agents in the 1990s (see Chapter 16). The formalization of reinforcement learning within a decision-theoretic framework also contributed to this shift (Sutton, 1988). Somewhat remarkably, almost all AI research until very recently has assumed that the performance measure can be exactly and correctly specified in the form of a utility function or reward function (Hadfield-Menell *et al.*, 2017a; Russell, 2019).

The general design for learning agents portrayed in Figure 2.15 is classic in the machine learning literature (Buchanan *et al.*, 1978; Mitchell, 1997). Examples of the design, as embodied in programs, go back at least as far as Arthur Samuel's (1959, 1967) learning program for playing checkers. Learning agents are discussed in depth in Chapters 19–22.

Some early papers on agent-based approaches are collected by Huhns and Singh (1998) and Wooldridge and Rao (1999). Texts on multiagent systems provide a good introduction to many aspects of agent design (Weiss, 2000a; Wooldridge, 2009). Several conference series devoted to agents began in the 1990s, including the International Workshop on Agent Theories, Architectures, and Languages (ATAL), the International Conference on Autonomous Agents (AGENTS), and the International Conference on Multi-Agent Systems (ICMAS). In 2002, these three merged to form the International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS). From 2000 to 2012 there were annual workshops on Agent-Oriented Software Engineering (AOSE). The journal *Autonomous Agents and Multi-Agent Systems* was founded in 1998. Finally, *Dung Beetle Ecology* (Hanski and Cambefort, 1991) provides a wealth of interesting information on the behavior of dung beetles. YouTube has inspiring video recordings of their activities.